CHAPTER 16

INHERITANCE: REUSABILITY AND EXTENDABILITY

The world around us is made of objects that share many similarities. These similarities can be classified into common groups. For example, in biology taxonomy organisms are classified into a hierarchy of group, from general to specific. In C++, objects are created from classes; and a class may share (inherit) some common data members and member functions that belong to another class or classes. As a result, a new class can be created based on an existing class rather than creating it from scratch. An inherited class can have its own data and function members and can modify or override the inherited members. Inheritance is an important tool of Object-Oriented Programming (OOP), because it promotes reusability and ease of extensibility by building on what is already there and customizing it as desired. A programmer can build a hierarchy that goes from a general class to a specific class by incorporating inheritance. With class hierarchy a program is easier to follow, debug, and modify. With inheritance, a class is built on an existing class that has already been tested; therefore, inheritance reduces the time and the cost of development as well as minimizes errors in the program. Inheritance is also known as ***derivation*** or ***specialization***. In fact, inheritance is not something new. For example, humans have organized knowledge into hierarchical structures such as the animal kingdoms.

CLASS INHERITANCE: GENERAL SYNTAX

The general form of a class inherited from another class is shown in fig 9.1. The access specifier (access control) can be **public, private** or **protected**. The syntax for class inheritance is the same as a regular class except that after the class name (derivedname) there is a single colon, the derivation access type (accessspecifier), and the name of the class (baseclassname) from which it is derived.

```
class derivedname : accessspecifier  baseclassname{
    accessspecifier: memberdata;
    accessspecifier: memberfunctions; };
```

Figure 16.1 – Class inheritance syntax

EXAMPLE OF INHERITANCE

Take a moment and categorize yourself as an object. For example, are you a full time employee, a part time student, or both? Do you have a bank account? Do you own a car? As human beings we categorize the objects around us into hierarchical structures. For example, livings things are divided into five kingdoms (plant, animal, fungi, etc.). Furthermore, these kingdoms are divided into smaller subcategories. Humans belong to the kingdom Animilia, the phylum Chordata, and the class Mammilia. These classes all share some commonality such as attributes (data) and behaviors (functions) that can be factored out.

EMPLOYEE CLASS: An employee can belong to a *base class* of person and several derived classes such as a salaried or hourly paid class both of which inherit from the person class. An employee can be full time or part time. Moreover, an employee can be a consultant, manager, or executive.

STUDENT CLASS: There may be different kinds of students in a college that all share common characteristics.
The student class is the *base class* and students can be further categorized into the following classes:

Undergraduate
Graduate
Full time
Part time
Freshman, sophomore, junior, senior
Exchange
International student

For example, Jane Doe is a part time, freshman, undergraduate, exchange student.

EXAMPLE OF INHERITANCE: COMPUTE AREA OF CIRCLE

The following program computes the area of a circle by providing the x and y coordinates of two given points. The program finds the radius using the distance formula. The program begins by defining the class **point** and class **circle**. The **circle** class inherits from the **point** class. The class **point** has two member functions: **setpoints( )**, which initializes the coordinates, and **distance( )**, which computes the distance between the given the points. The class **circle** inherits the **distance( )** function in order to compute the area.

```
#include <iostream.h>
#include <math.h>
class point{
public: int xstartpoint,ystartpoint;
        int xendpoint, yendpoint;
        void setpoints(int x1, int x2, int y1, int y2){
                xstartpoint = x1, xendpoint = x2;
                ystartpoint = y1, yendpoint = y2;}
        double distance(){
                return  sqrt((pow(xendpoint-xstartpoint,2))+(pow(yendpoint-ystartpoint,2)));}
    };//POINT
class circle: public point{
public: double radius;
        double area(){
                radius = distance() ; //function belongs to base class
                return radius * radius * 3.14;} };//CIRCLE
void main(){
        int x1coord = 2, y1coord = 12, x2coord= 6,y2coord = 15;
        circle mycircle;
        mycircle.setpoints (x1coord,x2coord,y1coord,y2coord);
        cout<<"CIRCLE AREA IS "<<mycircle.area()<<endl;    }//MAIN
```

Figure 16.2a – A program showing single inheritance

```
CIRCLE AREA IS 78.5
```

Figure 16.2b – Output of 16.2a

BASE CLASS, DERIVED CLASS-GENERALIZATION TO SPECIALIZATION

In building inheritance, the existing class is called the ***base class*** and the class that inherits from the base class is called the ***derived class.*** In the above example class **point** is the base class and class **circle** is the derived class. To determine a base class we factor out the common attributes (data) and behaviors (functions) from other classes. The factoring can continue until you get to a specific class and you want to focus and instantiate (create) an object. This leads from generalization to specialization.

```
class D : public B {
                        //.....
                        //.....
```

In the preceding example, **D** is the ***derived class*** (or ***subclass***) and **B** is the ***base class*** (or ***super class***). Note that class **D** inherits the data and functions from class **B**, therefore class **B** must exist before **D** can inherit from it.

PROTECTED MEMBERS INSTEAD OF PRIVATE

Recall, the access controls for a member class are: **private, public**, and **protected**. In the base class, instead of private, the keyword **protected** is used so that the subclasses can access it while not making it public to every other class. When an access control is specified as protected in the base class, it suggests that there will be a derived class (subclass) whose members can access the base class members as if it were private only to that derived class. Remember that in a class, the private members are only accessible directly by its members; however, the **friend** of the class, though not a member of the class, can access the private members. Friends of classes will be further discussed. However, by making a member of a class protected, its derived class can also access the protected member, as it is private to both the base class and the derived class.

```
#include <iostream.h>
class B {
protected: int bm; };

class D: public B {
public: accessbm(int x){ bm=x+2;} };

void main(){
        D ob;
        cout<<"bm VALUE IS "<<ob.accessbm(5)<<endl;
        }//MAIN
```

Figure 16.3a – Protected base class member example

```
bm VALUE IS 7
```

Figure 16.3b – Output of 16.3a

In the above example, what would happen if the access control became private instead of protected? The data member **bm** would be inaccessible.

DEFAULT INHERITANCE ACCESS TYPE: PRIVATE

The default inheritance access for a derived class is private, which means the non-private base class members become private for the derived class; note that the private members of the base class are not accessible in the derived class. Public and protected members of the base class become accessible by the derived class member functions when the derived access modifier is private. However, once the public and protected members of the base class become private in the derived class, these members are inaccessible to the outsider. This means that if you create an object of the derived class in the main program, it cannot access any of the members in the class. Private inheritance, though it is the default, might seem useless. Setting the inheritance access type (control) to anything other than public,

such as private or protected, will downgrade the accessibility of the base class members. For example, if the inheritance access type is private and the access type of the base class member is public, it forces it to become private and therefore accessible only to the members of the derived class. Note that just because private members of the base class are accessible inside the derived class, it does not mean that objects of the derived class can access the members of the base class directly. The members are only accessible through the public utility functions in the derived class or friend. To summarize and settle the confusion when dealing with members of the base class in the derived class, visualize the derived class as a new class in which the base class members are included in the derived class (except private base members) but now with new access permission types from the impact of the derived class access modifier. For example, in the following program, think that the protected member **bm** in class **B** is physically present in the derived class **D** as private **bm**.

```
#include <iostream.h>
class B {
protected: int bm; };

class DPV: private B {
public: accessbm(int x){ bm=x+2;} };

void main(){
        DPV ob;
        cout<<"bm VALUE IS "<<ob.accessbm(5)<<endl;
        }//MAIN
```

Figure 16.4a – Private inheritance example

```
bm VALUE IS 7
```

Figure 16.4b – Output of 16.4a

PRIVATE INHERITANCE

In the following example, the class **DPV** is privately derived from the base class **B**, making the base members that are public and protected to private. Therefore, the members of the derived class can access these inherited members. However, since these inherited members become private in the class **DPV**, they are no longer accessible to any other classes or any classes derived from **DPV.**

```
#include <iostream.h>
class B {
protected: int bm; };

class DPV: private B {
public: accessbm(int x){ bm=x+2;} };

class DDPV: public DPV {
public: accessbmdpv(int x){accessbm(x); } };

void main(){
        DDPV ob;
        cout<<"bm VALUE IS "<<ob.accessbmdpv(5)<<endl;
        }//MAIN
```

Figure 16.5a – Private inheritance example

```
bm VALUE IS 7
```

Figure 16.5b – Output of 16.5a

DERIVATION TYPE (INHERITANCE ACCESS): PUBLIC OR PROTECTED

How does a derived class access the base class members when the default inheritance access is private? A derived class member can only access the public and protected members of the base class. Therefore, the inheritance access has to be either public or protected for the derived class to access base class members. It is common practice to have inheritance access as public despite its default private access.

INHERITANCE ACCESS CONTROL (MODIFIER)

An inheritance access type modifier or access specifier can be private, protected, or public. Each of these access modifiers determines the accessibility of the derived members as well as how the derived members can access the base members. A private modifier will force the access to become private, while a protected modifier will force the access to become protected. However, the public modifier leaves the access as it is set in the base class. In another words, the modifier can make the access more restricted than defined in the base class. Keep in mind, members of the base class with private access become inaccessible in derived classes. By default the inheritance access is private forcing all members of the base class to be treated as private in the derived class and as a result, the other classes and the program cannot to access them.

```
class B  {
public:  //...
private: //...
protected: //...  };

class D: public B {
public: //...
private: //...
protected: //...  };
```

If data or functions are declared as public in a base class and the base class access is defined as public in a derived class, its access is public in the derived class. In summary, if the base class is inherited as public the access in the derived class will be the same. If the base class is inherited as protected, it will force the access in the derived class to become protected. And finally, if the base class is inherited as private, it forces the access in the derived class to become private.

USING CONSTRUCTORS WITH INHERITANCE

In a class, a constructor is used to initialize the data members. A base class and a derived class can have their own constructors. When an object of a derived class is created, which constructor will be called first? The constructor for the base class is called first and then the constructor for the derived class will follow.

```
#include <iostream.h>
class B {
public:
        B() { cout<<"BASE CLASS CONSTRUCTOR IS CALLED"<<endl; }  };//B

class  D: public B {
public:
        D(){ cout<<"DERIVED CLASS CONSTRUCTOR IS CALLED"<<endl; } };//D

void main(){
        D dob;
        }//MAIN
```

Figure 16.6a – Constructors with inheritance

```
BASE CLASS CONSTRUCTOR IS CALLED
DERIVED CLASS CONSTRUCTOR IS CALLED
```

Figure 16.6b – Output of 16.6a

USING DESTRUCTORS WITH INHERITANCE

In a class, a destructor is used to de-initialize the data members. The destructor for the derived class is called first and then the destructor for the base class will follow. The order in which the destructors of derived class and base class are called is the opposite of constructors.

```
#include <iostream.h>
class B {
public:
        ~ B() { cout<<"BASE CLASS DESTRUCTOR IS CALLED"<<endl; }  };//B

class  D: public B {
 public:
        ~ D(){ cout<<"DERIVED CLASS DESTRUCTOR IS CALLED"<<endl; } };//D

 void main(){
        D dob;
        }//MAIN
```

Figure 16.7a – Destructors with inheritance

```
DERIVED CLASS DESTRUCTOR IS CALLED
BASE CLASS DESTRUCTOR IS CALLED
```

Figure 16.7b – Output of 16.7a

MULTIPLE INHERITANCE

Like a child that inherits from both parents, a derived class in a multiple inheritance can have more than one base class. Most inheritance in C++ is single inheritance with a derived class inheriting from one base class. In several situations, it is desirable to use multiple inheritances. In fact we have been using multiple inheritance in our programs since we started learning how to program in C++. For example, whenever we use **#include <iostream.h>**, we are indirectly using multiple inheritance. The class iostream is inherited from two base classes: **istream** as well as **ostream.** One example of multiple inheritance is to create a new class such as studentemployee that inherits from both the student class and the employee class. Despite its power many object-oriented programming languages such as Java do not support multiple inheritance. There are ambiguities with multiple inheritance and casting a pointer from a subclass to a base class can create confusion for a programmer as well as more work for a compiler writer. Figure 16.8 is a program that demonstrates multiple inheritance; the derived class D inherits from two base classes, B1 and B2.

```
#include<iostream.h>
class B1{
public:   int x;
          B1(){}
          B1(int a){x=a;}
          int getx(){ return x;}  };//B1
class B2{
public:   int y;
          B2(){}
         B2(int b){y=b;}
          int gety(){ return y;}  };//B2
class D: public B1, public B2{
public:   int z;
         D(int a, int b, int c):B1(a),B2(b){
                              z=c;}
        int getz(){return z;}  };//D
void main(){
        D ob(1,2,3);
        cout<<ob.getx()<<endl;
        cout<<ob.gety()<<endl;
        cout<<ob.getz()<<endl;
        }//MAIN
```

Figure 16.8a – Multiple inheritance example

```
1
2
3
```

Figure 16.8b – Output of 16.8a

The next example uses multiple inheritance to relate to parents and a child. The two base classes are the father and mother classes and the child class inherits from both. The program tries to guess the child's eye color based on the eye color of the parents.

```
#include<iostream.h>
class mother{
public:
        int eye;
        mother(){eye=0;}
        mother(int x){eye=x;}
        int getmothereye(){return eye;}
};//MOTHER
class father{
public:
        int eye;
        father(){eye=0;}
        father(int y){eye=y;}
        int getfathereye(){ return eye;}
};//FATHER
class child: public mother,public father{
public:  int eye;
        child(){eye=0;}
        child(int a,int b):mother(a),father(b){
                        if(a==b)
                                eye=a;
                        else
                                eye=2;}
                int childeye(){return eye;}
};//CHILD
void main(){
        int mcolor, fcolor;
        cout<<"Enter the color of mother's eyes."<<endl;
        cout<<"(1 for blue, 2 for brown): ";
        cin>>mcolor;
        cout<<"Enter the color of father's eyes."<<endl;
        cout<<"(1 for blue, 2 for brown): ";
        cin>>fcolor;
        child harry(mcolor, fcolor);
        cout<<"Mother's eyes: "<<harry.getmothereye()<<endl;
        cout<<"Father's eyes: "<<harry.getfathereye()<<endl;
        cout<<"Child's eyes: "<<harry.childeye()<<endl;
}//MAIN
```

Figure 16.9a – Determining eye color using multiple inheritance

```
Enter the color of mother's eyes.
(1 for blue, 2 for brown): 1
Enter the color of father's eyes.
(1 for blue, 2 for brown): 2
Mother's eyes: 1
Father's eyes: 2
Child's eyes: 2
```

Figure 16.9b – Output of 16.9a

VIRTUAL FUNCTION

A virtual function is a member function that is defined in one class (the base class) and will be redefined by subsequent classes (derived classes). In order to create a virtual function, precede the function's declaration with the keyword **virtual** in the base class. The subsequent classes (derived classes) can implement the virtual function depending on the object they are dealing with, and there is no need for the keyword virtual. In classes with virtual functions, the function name is the same but there are several different implementations- one interface with many implementations is the concept of polymorphism. Another kind of polymorphism can be established during run-time is to use a pointer to the base class, this is known as run-time polymorphism. To assign the address of an object of a derived class to a pointer pointing to the base class is the essence of polymorphism. For example, you may want to send a different message in each derived class.

PURE VIRTUAL FUNCTION

A pure virtual function is a function that is in an abstract base class and does not have implementation in its class and instead its body has =0 like an assignment, as shown below. A pure virtual function must be preceded by the keyword **virtual**, however, this is optional in its subclasses, but it is a good practice to include the word **virtual** in the declaration.

```
class employee {
        virtual double computesalary()=0; };
```

Different versions of the pure virtual function's implementation are written in the subclasses. The following example computes the area of a given shape. Whether the area is computed for a circle, square, or rectangle depends on the object that calls the function in the main program.

```
#include<iostream.h>
class shapearea{
 public:   virtual int computearea()=0;
        };//SHAPEAREA
class circle: public shapearea {
 private:  int radius;
 public:   int computearea(){
                radius=20;
                cout<<"The area of the circle is: "<<3.14*radius*radius<<endl;
                return 0; }
        };//CIRCLE
class square: public shapearea{
 private: int side;
 public:  int computearea(){
                 side=12;
                 cout<<"The area of the square is: "<<side*side<<endl;
                 return 0; }
        };//SQUARE
class rectangle: public shapearea{
private:  int length;
        int width;
public:   int computearea(){
                length=5;
                width=10;
                cout<<"The area of the rectangle is: "<<length*width<<endl;
                return 0; }
        };//RECTANGLE
void main(){
        circle a;
        square b;
        rectangle c;
        a.computearea();
        b.computearea();
        c.computearea();
        }//MAIN
```

Figure 16.10a – Pure Virtual Function Example

```
The area of the circle is: 1256
The area of the square is: 144
The area of the rectangle is: 50
```

Figure 16.10b – Output of 16.10a

OVERRIDING BASE CLASS MEMBERS

In a derived class, you can modify or change the way a function of a class works by redefining it; this is called overriding a base class function. If the function is a virtual function it is known as overriding while for a non-virtual function, the redefinition is called redefining.

EARLY BINDING VERSUS LATE BINDING

The association of the function's name with the starting address of the function's code (entry point) is known as binding. A function's name should bind to the code that implements the function. Early binding is also known as compile-time binding, when the compiler determines which function (code) will be executed. An alternative to early binding is late binding, or run-time binding. In late binding the function to be executed is determined during run time, not during compile time.

RUN-TIME POLYMORPHISM

Polymorphism means many forms, using one name throughout the program for a function but this function has several definitions. In other words, associating multiple meanings to one function name. Run-time polymorphism (late binding) is when a function to be called finds its associated definition during the program execution rather than normal compiling time. For runtime polymorphism, a different version of a virtual function may be used in a base or other derived classes. In addition, a pointer or a reference to a base class is used to refer to each virtual function after being assigned by the object.

```
#include <iostream.h>
class B {
public: virtual vm( ){cout <<"HERE IS THE BASE FUNCTION "<<endl;}
        };//B
class D : public B {
public: virtual  vm( ){ cout <<"HERE IS THE DERIVED FUNCTION "<<endl;}
        };//D
void main(){
        B obb;
        D obd;
        B* obp;
        obp= &obb;
        obp->vm();
        obp=&obd;
        obp->vm();
        }//MAIN
```

Figure 16.11a – Program to demonstrate run-time polymorphism

```
HERE IS THE BASE FUNCTION
HERE IS THE DERIVED FUNCTION
```

Figure 16.11b – Output of 16.11a

ABSTRACT CLASS AND PURE VIRTUAL FUNCTIONS

A class with at least one pure virtual function is known as an ***abstract base class.*** A ***pure virtual function*** does not have implementation in its class but will be overridden at later time in a derived class. It is important to know that you cannot create an object from the abstract base class.

FRIEND ACCESS RIGHTS AND INHERITANCE

A class can make all of its private members accessible to another class by a friend declaration. A friend function is not a member of the class but has access to the class's members. The friend declaration is placed inside the class and is preceded by the keyword **friend** and its definition is treated as a regular non-member function. In the friend function definition the keyword **friend** does not has to be included. Friend functions may violate the concept of encapsulation. Just to recall, in inheritance the private members of a base class are not accessible by the derived class.

WHAT WOULD NOT BE INHERITED

A derived class inherits all members except the friend functions and the overloaded operators. The constructor, destructor, copy constructor, and assignment operator of the base class are not inherited but can be used by the derived classes in conjunction with their own.

WHERE DOES ABSTRACTION COME FROM

The word abstraction comes from the Latin word ***abs*** meaning "***away from***" and the word ***trahere*** meaning to "***draw***". The idea is to take away the unnecessary detailed characteristics and reduce it to a form so that it can be used efficiently with less complexity, yet is identifiable. In programming, abstraction is a word often used these days. It is related to encapsulation, meaning putting data and its related functionality together under a name (known as class). Abstraction hides data (details) and restrains unwanted access. This will emphasize the separation of implementation from the interface. The notion of Abstract Data Type (ADT) is to build user defined types in a similar way as system data types which encompasses necessary data and operations and from which objects are instantiated and then used. For example, an employee class that is a user-defined type should behave the same way as built in types such as int (integer).

IS-A RELATIONSHIP AND HAS-A RELATIONSHIP

When a class inherits another class, the relationship is **is-a**. For example, when having a student class and a part-time student class that inherits the student class, the relationship is **is-a**. Inheritance is based on is-a relationship; a part time student **is a** student and similarly a salaried employee **is an** employee. When a class is nested inside another class

the relationship would be **has-a**. For example, there are two classes such as class employee and class date, and the class date is a member of the class employee.

TOPICS NOT COVERED HERE

There are certain member functions such as overloaded operators that cannot be inherited. Multiple inheritance can pose its own problems as to which base class the derived members belong to especially if there are generations of inheritance. Similarly, the constructor initialization of a base class whether there is a default constructor or not can become complicated. Conversion of one type to another can also become troublesome. A constructor cannot be virtual but a destructor can be virtual. Assigning one object of a inherited class to another can pose a problem due to type compatibility and, instead, pointer assignment may be a solution. Many people argued that single inheritance is sufficient and that multiple inheritance will cause errors for programmers due to the ambiguity of the members and the complexity in regards to initializing the objects by their constructors and cleaning them up by destructors. However, Microsoft's Foundation Classes (MFC) use multiple inheritance.

CLOSING REMARKS AND LOOKING AHEAD

Inheritance is a powerful feature of Object-Oriented-Programming that promotes code reusability and extensibility. By inheritance a new class can be created by extending the existing classes and by doing so, you do not reinvent the code but rather you reuse the proven and tested code which in a long run will saves time and cost. In addition, in a complex situation an object can be categorized by a hierarchical classification, progressing from a general class to a specific class. There are three different types of derivations: public, protected, and private. The choice of derivation indicates how the derived class receives the members of the base class. A private member of a base class is inaccessible to the derived class but a function of a derived class can call the non-private functions of base class to access the private data members.

A new class can be derived from one base class (single inheritance) or multiple base classes (multiple inheritance). Before a program is written the hierarchy of classes should be set, this includes the relationship of classes and sharing of the data and member functions. However, it is a common practice to reorganize a class hierarchy by factoring out the common parts of the two classes by creating a new base class and two subclasses. Inheritance can become intense when an object of one class, such as base class, is used instead of an object of the derived classes, or vice versa. Keep in mind that the derived classes have more than what the base class has and as a result a conversion becomes necessary. Moreover, by using the base class pointer it is possible to point to the derived classes. Doing so enables a programmer to access a collection of objects of different classes.